View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by DBIS EPub

Flexibility for Distributed Workflows*

Manfred Reichert[1,2], Thomas Bauer[3], Peter Dadam[1]

[1] Institute of Databases and Information Systems, University of Ulm, Germany
[2] Information Systems Group, University of Twente, The Netherlands
[3] Daimler AG, Group Research & Adv. Engineering, GR/EPD, Germany

Abstract

This chapter shows how flexibility can be realized for distributed workflows. The capability to dynamically adapt workflow instances during runtime (e.g., to add, delete or move activities) constitutes a fundamental challenge for any workflow management system (WfMS). While there has been significant research on ad-hoc workflow changes and on related correctness issues, there exists only little work on how to provide respective runtime flexibility in an enterprise-wide context as well. Here, scalability at the presence of high loads constitutes an essential requirement, often necessitating distributed (i.e., piecewise) control of a workflow instance by different workflow servers, which should be as independent from each other as possible. This chapter presents advanced concepts and techniques for enabling ad-hoc workflow changes in a distributed WfMS as well. Our focus is on minimizing the communication costs among workflow servers, while ensuring a correct execution behavior as well as correctness of ad-hoc workflow changes at any time.

INTRODUCTION

For a variety of reasons enterprises are developing a growing interest in aligning their information systems such that they become process-aware (Lenz, 2007; Müller, 2006; Mutschler 2006; Mutschler, 2008a). Such process-aware information systems (PAISs) offer the right *tasks* at the right *point in time* to the right *actors* along with the *information*, *resources* and *application services* needed to perform these tasks (Dadam, 2000). Business process management technology offers promising perspectives to achieve this goal (Weske, 2007). Examples include workflow management systems and case handling tools (Günther, 2008 a; Mutschler, 2008b).

A workflow management system (WfMS) enables computer-supported business processes (i.e., *workflows*) to be executed in a distributed system environment (Bauer, 1999; Muth, 1998; Shegalov, 2001). Usually, a WfMS provides powerful tools for implementing enterprise-wide, process-aware information systems (PAISs) (Dadam,

* Corresponding author: Prof. Dr. Manfred Reichert, University of Ulm, Faculty of Engineering and Computer Science, Institute of Databases and Information Systems, Oberer Eselsberg, 89069 Ulm, GERMANY; E-Mail: manfred.reichert@uni-ulm.de

1999). As opposed to data- or function-centered information systems, a WfMS separates the specification of the process logic (i.e., the control and data flow between the process activities) from application coding (Dadam, 2000; Weber, 2007); i.e., process logic can be described explicitly in terms of a *workflow template* providing the schema for *workflow enactment* (workflow schema for short). The different *activities*, in turn, are implemented as loosely coupled *application services* that can expect that their input parameters are provided upon invocation by the WfMS and which only have to produce correct values for their output parameters. Usually, the core of the *workflow layer* is built by the WfMS which provides generic functions for modeling, configuring, executing, and monitoring workflows.

This separation of concerns increases maintainability and reduces cost of change (Mutschler, 2008a; Weber, 2008a); i.e., changes to one layer often can be performed without affecting other layers; e.g., changing the execution order of workflow (WF) activities or adding new activities to a *WF schema* can, to a large degree, be accomplished without touching any of the associated application services (Dadam et al., 2000). Furthermore, a *WF schema* can be checked for the absence of flaws already at buildtime; i.e., deadlocks, livelocks and faulty data flow specifications (van der Aalst, 2000; Reichert, 1998a) can be excluded in an early stage of the process lifecycle (Weber, 2009; Weber, 2006a). At run-time, new *WF instances* can be created and executed according to the underlying *WF schema*. When an activity becomes activated, a respective *work item* is assigned to the *worklists* of authorized users (which are determined based on the *actor assignment* associated with the corresponding activity). One example of such a WfMS constitutes the ADEPT system we have developed during the last years (Reichert, 2003c).

Problem Statement

A centralized WfMS shows deficits when being confronted with high loads or when supporting cross-departmental processes (Reichert, 1999; Dadam, 2000). In the ADEPT project, we have considered this by realizing a *distributed WfMS* made up of several WF servers (Bauer, 1997; Bauer, 1999; Bauer, 2003; Montagut, 2007). In this distributed variant of the ADEPT system, we allow WF designers to subdivide a WF schema into several partitions which are then controlled ”piecewise” by different WF servers in order to obtain favorable communication behavior. Note that similar approaches have been discussed in literature (Alonso, 1995; Casati, 1996; Cichocki, 2000; Dogac, 1997; Gronemann, 1999; Guth, 1998; Kochut, 2003; Muth, 1998; Schuster, 1999; Sheth, 1997; Weske, 1999).

Comparable to centralized WfMS, also a distributed WfMS needs to be flexible to cover the broad spectrum of processes we can find in today’s organizations (Bassil, 2004; Kochut, 2003; Lenz, 2007; Minor, 2007; Müller, 2006; Reichert, 1998 b). Thus, at the WF instance level it should be possible to flexibly deviate from the predefined WF schema during runtime. As reported in literature (van der Aalst, 2001a; Pesic, 2007, Reichert, 1998a; Mourào, 2007; Weber 2006a) such ad-hoc workflow changes become necessary to deal with exceptional and changing situations. Within the ADEPT project we developed an advanced technology for the support of such ad-hoc changes (Reichert, 1998a; Reichert, 2003a; Reichert, 2003b). In particular, ADEPT allows authorized users (or agents) to dynamically modify running WF instances, but without causing run-time errors or inconsistencies in the sequel (Rinderle, 2003).

In our previous work we considered distributed execution of a partitioned WF schema and ad-hoc WF changes as separate issues (e.g., Reichert, 1998; Bauer, 2003). In fact, we did not systematically examine how these two fundamental aspects of a large-scale WfMS interact with each other. Obviously, integrated support of respective features is by far not trivial as their goals are different. The support of ad-hoc WF changes and the correct processing of the WF instances afterwards prescribe a logically central control instance (i.e., a logically central WF server) to ensure correctness (Reichert, 1998a). This, however, contradicts to the accomplishments achieved by distributed WF execution (Bauer, 1997; Bauer, 2000). Note that one central WF server always decreases WfMS availability and increases communication costs between WF clients and WF server (Kamath, 1996). One reason for this lies in the fact that a central control engine must be informed of all changes concerning the state of a WF instance. In particular, information on instance states is needed to decide whether an intended ad-hoc change is applicable in a given context; i.e., whether the considered WF instance is compliant with the resulting WF schema (Reichert, 1998a; Rinderle, 2004a; Rinderle-Ma, 2008a).

Contribution

This chapter provides an extended version of the work we presented in (Reichert, 2007). It describes an approach which enables ad-hoc changes of single WF instances in a *distributed WfMS*; i.e., a WfMS with *WF schema partitioning* and *distributed WF control*. As a prerequisite, distributed WF control must not affect applicability of ad-hoc changes; i.e., each change, which is allowed for the central case, should be applicable in the context of distributed WF execution as well. The support of such ad-hoc changes, in turn, must not impact distributed WF control. In particular, distributed WF execution should not necessitate a great deal of additional communication effort due to the introduction of WF instance changes. Finally, ad-hoc changes should be correctly performed and as efficiently as possible.

To deal with these requirements it is crucial to identify the WF servers of the distributed WfMS to be involved in the synchronization of an ad-hoc change. Most likely we have to consider those WF servers currently executing the respective WF instance. These *active servers* need to know the schema and state of a changed WF instance in order to correctly control its execution afterwards. We need an efficient approach for determining the set of active servers controlling a particular WF instance. This must be possible without a substantial expense of communication efforts. In addition, we have to decide whether, when and how a changed WF instance schema has to be transmitted to other WF servers. As essential requirement the amount of communication should not exceed acceptable limits.

This chapter is structured as follows: We first give background information needed for the further understanding and we introduce basic issues related to distributed WF execution as accomplished in the ADEPT approach. Following this, we first describe how ad-hoc instance changes can be performed in the distributed variant of the ADEPT WfMS. Then we show how individually modified WF instances can be efficiently executed in such distributed WfMS. Finally, we describe our proof-of-concept prototype and discuss related work. The chapter concludes with a summary and outlook.

BACKGROUNDS

We first show how workflows can be modeled in the ADEPT WfMS. Following this we discuss fundamental issues related to ad-hoc changes of single WF instances.

Workflow Modeling and Execution in ADEPT

When implementing a workflow in a PAIS its control and data flow has to be explicitly defined based on the modeling constructs provided by the used *WF meta model*. More precisely, for each business process to be supported, a *WF type* represented by a *WF schema* is defined. For one particular WF type several WF schemes may exist representing the different *versions* and the *evolution* of this WF type over time. Figure 1 shows a simple example of a WF schema as modeled in ADEPT. The depicted schema comprises seven activities connected through *control edges*. Generally, control edges specify precedence relations between the activities. For example, activity *order medical examination* is followed by activity *make appointment*, whereas activities *prepare patient* and *inform patient* may be executed in parallel. Furthermore, the WF schema contains a loop structure, which allows for the repetitive execution of the depicted WF fragment.

The *ADEPT WF meta model* allows for the integrated modeling of different WF aspects including activities, control and data flow, actor assignments, semantical constraints, and resources. Here we focus on the first three perspectives.

Control flow modeling. As depicted in Figure 1, the control flow of a WF schema is represented as attributed graph with distinguishable node and edge types. This allows for efficient correctness checks and eases the handling of loop backs. Formally, a *control flow schema* corresponds to a tuple (N,E, ...) with node set N and edge set E. Each control edge e ∈ E has one of the edge types CONTROL_E, SYNC_E or LOOP_E: CONTROL_E expresses a normal precedence relation, whereas SYNC_E allows to express a wait-for relation between activities of parallel branches (Reichert, 2000). Finally, LOOP_E represents a loop backward edge. Similarly, each node n ∈ N has one of the node types STARTFLOW, ENDFLOW, ACTIVITY, STARTLOOP, ENDLOOP, AND-/XOR-Split, and AND-/XOR-Join. Based on these elements, we can model sequences, parallel branchings, conditional branchings, and loop backs. ADEPT adopts concepts from block-structured process description languages, but enriches them by additional control structures. More precisely, branchings as well as loops have exactly one entry and one exit node. Furthermore, control blocks may be nested, but must not overlap. As this limits expressive power, in addition, the aforementioned synchronization edges can be used for process modeling (Reichert, 2000).

Data flow modeling. Data exchange between activities is realized through writing and reading *WF variables* (denoted as *data elements* in the following). Data elements are connected with input and output parameters of WF activities. Each input parameter of a particular activity is mapped to exactly one data element by a *read data edge* and each activity output parameter is connected to a data element by a *write data edge*. An example is depicted in Figure 1. Activity *order medical examination* writes data element *patientID* which is then read by subsequent activity *perform ex-*

amination. The total collection of *data elements* and *data edges* constitutes the *data flow schema*. For its modeling, a number of restrictions has to be met. The most important one ensures that all data elements mandatorily read by an activity X must have been written before X is started. In particular, this must be ensured independent from the execution path leading to activation of X.

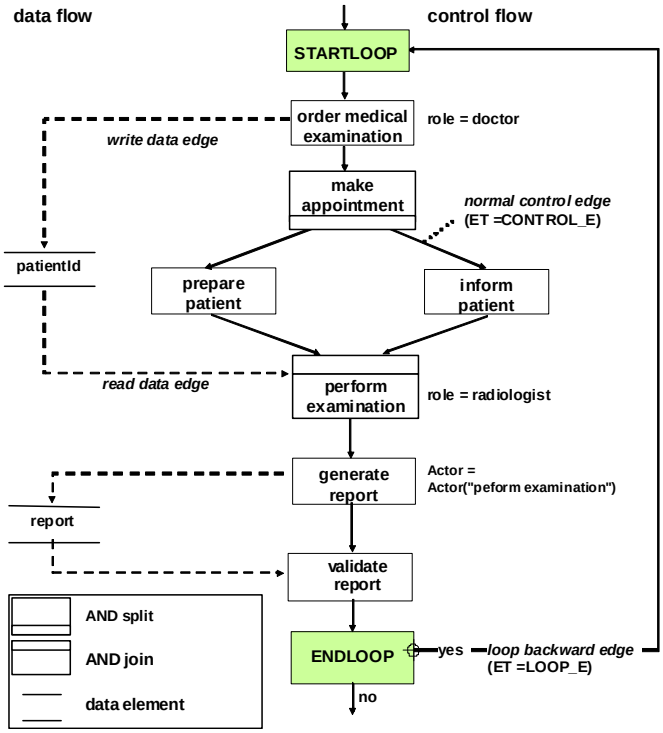


Figure 1 Example of a simple ADEPT WF schema

Based on a given *WF schema* new *WF instances* can be created and executed. ADEPT orchestrates them according to the defined control flow. Regarding a single activity, initially, its status is set to NOT_ACTIVATED. It changes to ACTIVATED when all preconditions for executing this activity are met. In this case corresponding work items are inserted into the worklists of authorized users. If one of them selects the respective item from his worklist, activity status changes to RUNNING and respective work items are removed from other worklists. Furthermore, the application service associated with the activity is started. At successful termination, activity status changes to COMPLETED. Generally, a large number of WF instances being in different states may run on a particular WF schema. To determine which activities are to be executed next, WF enactment in ADEPT is based on a well-defined operational semantics (Reichert, 1998a; Reichert, 2000). Furthermore, for each WF instance we maintain information about its current state by assigning respective markings to the nodes and edges of its WF schema. Figure 2 shows two WF instances running on the WF schema depicted in Figure 1.

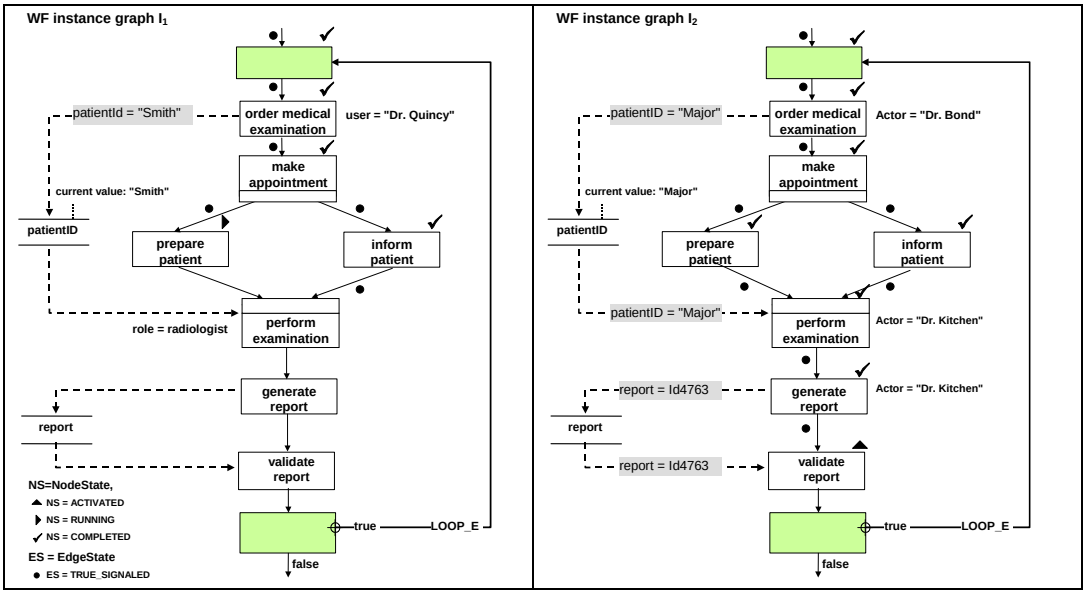


Figure 2: Examples of two WF instances running on the WF schema from Figure 1

Ad-hoc Workflow Changes in ADEPT

To allow users to flexibly react in exceptional situations and to dynamically evolve the structure of in-progress WF instances over time, ADEPT provides support for *ad-hoc changes*. Generally, WF flexibility can be achieved either through *structural adaptations* of WF schemes (Reichert 1998; Rinderle, 2004a; Rinderle, 2005) or by allowing for loosely specified WF schemes, which can be refined by users during runtime according to predefined criteria (Adams, 2006; Han, 1998; Sadiq 2001; Sadiq 2005; Weber, 2007). This chapter focuses on structural schema adaptations of single WF instances; i.e., *ad-hoc changes* which can be applied to single WF instances in order to cope with exceptional situations.

Usually, the introduction of ad-hoc changes results in an instance-specific WF schema (Reichert, 1998a), which we also denote as the *execution schema* of the instance in the following; i.e., change effects are instance-specific and do not affect any other WF instance. In a medical treatment process, for example, current medication of a particular patient might have to be discontinued due to an allergic reaction of this particular patient.

ADEPT provides a set of high-level *change operations* and *change patterns*, respectively, for realizing structural schema adaptations. In particular, respective change operations abstract from the concrete schema transformations becoming necessary to realize a particular change. Examples of ADEPT change operations include the insertion of a schema fragment between two activity sets or the movement of a fragment from its current position within a WF schema to a new one. Generally, change operations can be applied to the whole WF schema, i.e., the region to which the respective change operation is applied can be chosen dynamically (as opposed to late modeling approaches where changes are restricted to a predefined region). Therefore, the ADEPT change operations are suited for dealing with exceptions. Furthermore, it becomes possible to associate pre- and post-conditions with them. This, in turn, enables us to guarantee soundness when applying the respective change operations (Reichert, 1998a). Preserving soundness will be of particular importance if

ad-hoc changes are introduced by end users or – even more challenging – by software agents (Golani, 2006; Bassil, 2004).

We do not present all change patterns supported by ADEPT here, but only give three examples. For details on process change patterns as well as their formal semantics we refer to (Weber, 2007; Weber, 2008a; Rinderle-Ma, 2008b):

- *Insert fragment*: This operation can be used to add a schema fragment (i.e., a single activity or a complete block) to a WF schema. One parameterization describes the position at which the new fragment is embedded in the respective WF schema. For example, ADEPT allows to serially insert a fragment between two directly succeeding activities as well as to insert new fragments between two sets of activities meeting certain constraints. Special cases of the latter variant include the insertion of a fragment in parallel to another one (*parallel insert;* see Figure 3a) or the additional association of the newly added fragment with an execution condition (*conditional insert*).
- *Delete fragment*. This operation can be used to remove single activities or blocks.
- *Move Fragment*. This operation allows users to move a fragment from its current position to a new one. Like for *Insert Process Fragment*, an important parameterization specifies the way the fragment is re-embedded in the WF schema. Although the move operation could be realized by the combined use of the insert and delete operation, ADEPT introduces it as separate operation, since it provides a higher level of abstraction to users.

By the combined use of these and other change operations, complex schema adaptations can be realized at a high level of abstraction.

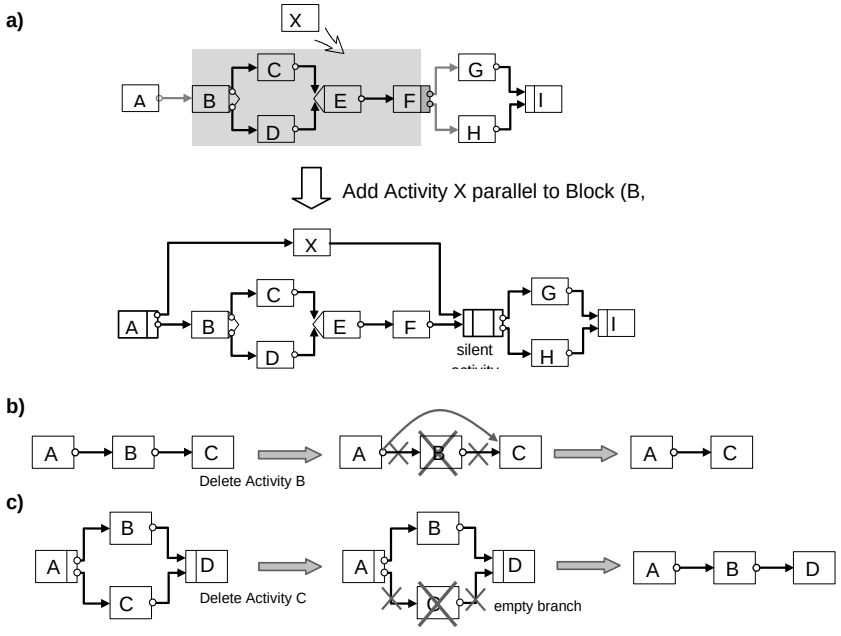


Figure 3. Insertion (a) and deletion (b+c) of process activities in ADEPT

So far, we have only considered structural issues. An example of an ad-hoc change applied at the WF instance level is shown in Figure 4. The depicted WF instance is modified by inserting new activity x in parallel to the existing activity b. Taking the user specification of the desired change ("insert activity x between a and c"), first of all, ADEPT checks whether this change can be applied; i.e., whether all correctness

properties guaranteed by formal checks at buildtime are further met. If this is the case, ADEPT automatically calculates the *basic schema transformations* (i.e., change primitives like *insert node* or *delete edge*) to be applied to the execution schema of the given WF instance. In addition, it determines the new state of the WF instance in order to correctly proceed with the flow of control afterwards. In our example the state of the newly inserted activity `x` is automatically set to `ACTIVATED`; i.e., the corresponding activity is immediately inserted into worklists of potential actors.

As illustrated in Figure 4 c, the required WF schema transformations (i.e., basic change primitives), together with the change specification, are recorded in the *change history* of the WF instance (Rinderle, 2006a). This history will be required, for example, if the WF instance has to be partially rolled back (Reichert, 2003a). Furthermore, ADEPT logs the occurrence of change events (and a reference to the corresponding change history entry) in the *execution history* of the WF instance as well. As example take the entry `DynModif(1)` in Figure 4 b which refers to the aforementioned ad-hoc change. Finally, the execution history contains other essential instance data, e.g., concerning the start and completion of activities.

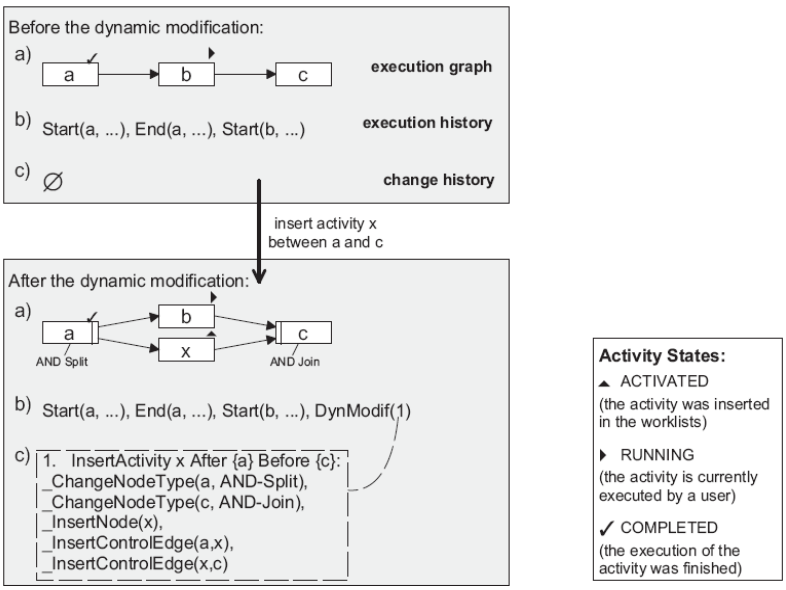


Figure 4. (Simplified) example of an ad-hoc instance change in a centralized WfMS with a) WF execution schema, b) execution history, and c) change history

Uncontrolled changes can lead to inconsistencies or errors. First of all, an ad-hoc change must result in a structurally correct WF instance schema. For example, deleting an activity can lead to missing input data for subsequent activities. This, in turn, can result in activity crashes or malfunctions when invoking the associated application service. Or, if a control edge is dynamically added without any checks, this can lead to deadlock-causing, cyclic dependencies. Besides structural soundness, we have to ensure that the respective WF instance is *compliant* with the modified WF schema (Casati 1998; Reichert, 1998; Rinderle, 2003; Rinderle, 2004a; Rinderle-Ma, 2008a); i.e., its execution log should be producible on the new WF schema as well. This will be not the case, if an activity is added to an already processed region of a WF schema. Generally, compliance is needed to avoid deadlocks or livelocks.

ADEPT precludes such errors and also ensures compliance. For this reason, formal pre- and post-conditions are defined for each change operation. They concern the state

as well as the structure of the WF instance. Before introducing an ad-hoc change, ADEPT analyzes whether it is permissible on the basis of the current state and structure of the WF instance; i.e., whether the defined pre- and post-conditions of the applied change operations can be met. Only if this is the case the structure and state of the WF execution graph will be modified accordingly. Regarding our example from Figure 4, for instance, it would not be allowed to delete the already completed activity a or to add a new activity as predecessor of a.

DISTRIBUTED WORKFLOW EXECUTION IN ADEPT

We investigated the requirements of enterprise-wide and cross-organizational WF-based applications in detail (Reichert, 1999). In the following we provide a brief summary of fundamental concepts we developed for distributed WF control. Though illustrations are based on ADEPT, the general principles behind them can be applied to other WfMS as well.

Usually, WfMS with one central WF server will be not adequate if the WF participants (i.e., the actors working on the activities) are distributed across multiple enterprises or organizational units. In such a case, the use of one central WF server will restrict the autonomy of the involved partners and be disadvantageous with respect to response times. Particularly, if organizations are widespread, response times will significantly increase due to long distance communications between WF clients and central WF server. In addition, owing to the large number of users and co-active WF instances typical for enterprise-wide applications, the WfMS is generally subjected to an extremely heavy load (Kamath, 1996; Sheth, 1997).

For these reasons, in the distributed variant of the ADEPT WfMS, a WF instance may be controlled by multiple WF servers; i.e., its schema may be partitioned at buildtime, and the partitions be controlled piecewise by the different WF servers during runtime (Bauer, 1997).[1] As soon as the execution of a partition completes, control over the WF instance is handed over to the next WF server. We denote the hand-over of the instance control from one WF server to another as *instance migration*.[2] An example is depicted in Figure 5.

When migrating a WF instance from one WF server to another, a description of its state has to be transferred to the target server before this server may take over control; i.e., before it may continue with instance execution. This includes, for example, information about the state of WF activities as well as WF relevant data; i.e., data elements connected with output parameters of activities. – To simplify matters, we assume that the WF templates (i.e., the WF type schemes) are replicated and stored on all relevant servers of the distributed WfMS.

To avoid unnecessary communication between WF servers, ADEPT allows to control parallel branches of an instance independent from each other – at least as long as no synchronization due to other reasons (e.g., ad-hoc changes) becomes necessary. In Figure 5 b, WF server s_3, which currently controls activity d, normally does not know

[1] To achieve better scalability, in ADEPT the same partition of different WF instances (with same type) can be controlled by multiple WF servers. Respective concepts, however, are outside the scope of this book chapter and are presented in (Bauer, 2003).

[2] In this context, migration should not be mixed up with the migration of a WF instance to a modified WF schema. Issues concerning the latter can be found in (Casati, 1998; Rinderle, 2004a-c).

how far execution has progressed in the upper branch (activities b and c). As advantage the WF servers controlling the activities of parallel branches do not need to be synchronized.

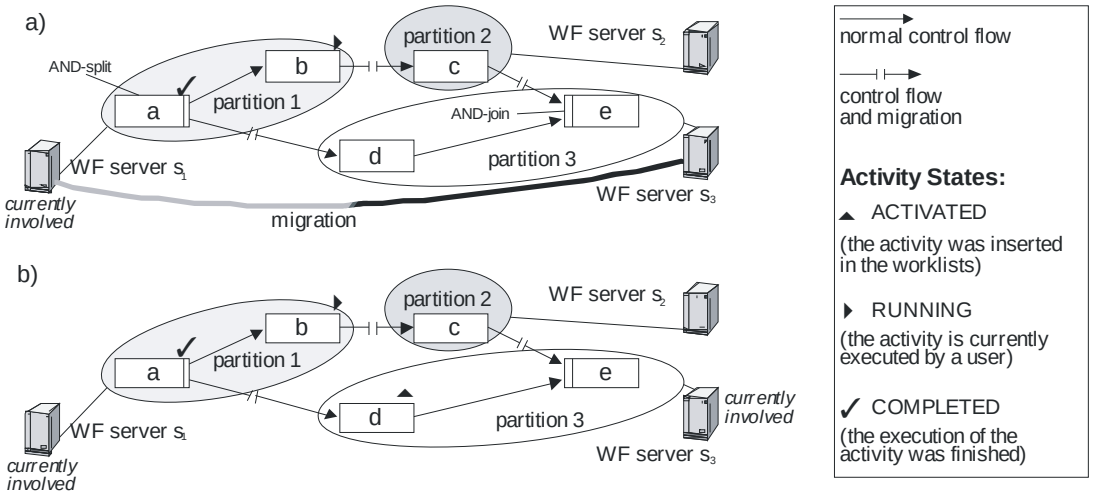


Figure 5. (a) Migration of a WF instance (from s_1 to s_3); (b) resulting state of the instance

The partitioning of WF schemes and distributed WF control have been successfully utilized in other approaches as well (Casati, 1996; Muth, 1998). In ADEPT, we also target at the minimization of communication costs. Concrete experiences we gained in working with commercial WfMS have shown that there is a great deal of communication between the WF server and its WF clients (e.g., displaying worklists), oftentimes necessitating the exchange of large amounts of data. This may result in an overloaded communication system.

Hence, the WF servers responsible for controlling activities in ADEPT are defined in such a way that communication in the overall system is reduced: Typically, the WF server controlling a particular activity is selected in a way such that it is located in the subnet to which most of the potential actors of the respective activity belong. (Bauer, 1997) describes respective algorithms. This way of selecting the server contributes to avoid cross-subnet communication between the WF server and its WF clients. Further benefits are improved response times and increased availability. This is achieved since neither a gateway nor a WAN is interposed when executing activities. Finally, the efficiency of the described approach – with respect to WF server load and communication costs – has been proven by means of comprehensive simulations (Bauer, 1999).

Usually, WF servers are assigned to the activities of a WF schema already at buildtime. In some cases, however, this static approach is insufficient. Extensions will become necessary if *dependent actor assignments* exist; e.g., activity n may have to be performed by the same actor as preceding activity m. Consequently, the potential actors of activity n depend on the concrete actor who processes activity m. Since this set of prospective actors can only be determined at run-time, WF server assignment should be deferred to runtime as well. Then, a server in a suitable subnet can be selected; i.e., one that is most favorable for the actors defined. For this purpose, ADEPT supports *variable server assignments* (Bauer, 2000; Bauer, 1999); i.e., expressions like "server in subnet of the actor performing activity m" can be assigned to activities and then be evaluated at runtime. This allows for the dynamic selection of

the WF server, which shall control the respective activity instance. Finally, (Bauer, 2004) deals with dynamic changes of server assignments in distributed WfMS.

REALIZING AD-HOC CHANGES IN A DISTRIBUTED WFMS

In a distributed WfMS it should be possible to perform ad-hoc changes of single WF instances just as in a central WfMS: The WfMS has to check whether the change may be applied taking the current structure and state of the respective WF instance into account. If the ad-hoc change is applicable (i.e., the instance has not progressed too far), the corresponding schema transformations will be determined and the WF schema belonging to the WF instance be modified accordingly (including adaptations of the WF instance state if required).

In order to check whether an intended ad-hoc change of a distributed WF instance is valid, first of all, the distributed WfMS needs to know the global state of the WF instance (or at least relevant parts of it). In case of parallel execution branches, for example, this state information may be distributed over several WF servers. It then has to be retrieved from these WF servers when the change shall be applied. How WF instance data can be efficiently transferred between the servers of a distributed WfMS has been described in (Bauer, 2001).

In the following we present a method for determining the WF servers on which the state information, relevant for checking the applicability of a particular ad-hoc change, is located. In contrast to a central WfMS, generally, in a distributed WfMS it is not sufficient to modify the execution schema of the WF instance solely on that WF server which controls the ad-hoc change. Otherwise, errors or inconsistencies might occur since the other WF servers might use outdated schema and state information when controlling the respective WF instance. In the following we show which WF servers have to be involved in the change procedure and how corresponding change protocols look like in ADEPT.

Synchronizing Workflow Servers in the Context of Ad-hoc Changes

An authorized user may invoke an ad-hoc change on any WF server which currently controls the WF instance in question. Yet as a rule, this WF server alone will not always be able to correctly perform the change. If other WF servers currently control parallel branches of the respective instance, state information from these WF servers might be needed as well. In addition, the WF server initiating the ad-hoc change must ensure that the change is also considered for the execution schemes of the respective WF instance, being managed by these other WF servers. This becomes necessary to enable these servers to correctly proceed with the flow in the sequel (see below). A naive solution would be to involve all WF servers of the WfMS by a broadcast. However, this approach is impractical in most cases as it is excessively expensive. In addition, all server machines of the WfMS must be available before an ad-hoc change can be performed. We come up with three alternative approaches:

Alternative 1 (Synchronize all Servers Concerned by the WF Instance). Alternative 1 considers all WF servers of the distributed WfMS which controlled the respective WF instance in the past, which are currently controlling respective WF activities, or which will be involved in the execution of future activities. Though the effort involved in communication is greatly reduced when compared to the naive solution

mentioned above, it may still be unduly large. For example, communication with those WF servers which were involved in controlling the WF instance in the past, but which will not re-participate in future, is superfluous. They do not need to be synchronized any more and the state information managed by these WF servers has already been transferred in previous migrations.

Alternative 2 (Synchronize all Current and Future Servers of the WF Instance). To be able to control a WF instance, a WF server needs to know its current execution schema. This, in turn, requires knowledge of all ad-hoc changes performed so far. Therefore, a new ad-hoc change must be made public to those WF servers which are either currently active in controlling the WF instance or which will be involved in its control in future. Thus, it seems to make sense to synchronize exactly these WF servers. However, with this approach, problems arise in connection with conditional branches. For XOR-splits, which are evaluated in future, it cannot always be determined in advance which execution branch will be chosen. As different execution branches may be controlled by different WF servers, the set of relevant WF servers cannot be calculated immediately. Generally, it is only possible to determine those WF servers *potentially* be involved in the control of the WF instance in future.

The situation will become worse if variable server assignments are used. Then, for a given WF instance it is not possible to determine the WF servers that will be potentially involved in the execution of future activities. Note that runtime data of the WF instance, as required for evaluating WF server assignment expressions, may not even exist at this point in time; e.g., in Figure 6, during execution of activity `g`, the WF server of activity `j` cannot be determined since the actor responsible for activity `i` has not been fixed yet. Thus the system will not always be able to synchronize future servers of the WF instance when an ad-hoc change takes place. As these WF servers do not need to be informed about the change at this time (since they do not yet control the WF instance) we suggest another approach.

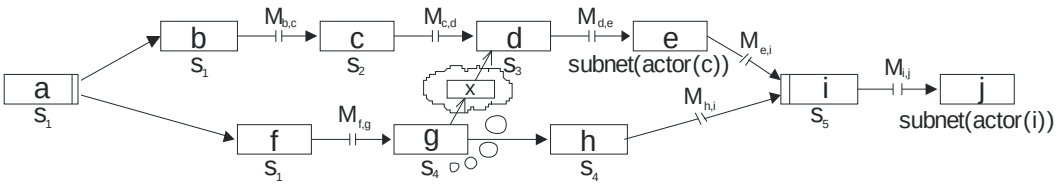


Figure 6. Insertion of activity `x` between activities `g` and `d` by server S_4.

Alternative 3 (Synchronize all Current Servers of the WF Instance). The only workable solution is to synchronize exclusively those WF servers currently involved in the control of the WF instance, i.e., the *active* WF servers. Generally, it is not trivial at all to determine which WF servers these in fact are. The reason is that in case of distributed WF control, for an active WF server of a WF instance, the execution state of the activities being executed in parallel (by other WF servers) is not known. As depicted in Figure 6, for example, WF server S_4, which controls activity `g`, does not know whether migration $M_{c,d}$ has already taken place and, as a result, whether the parallel branch is being controlled by WF server S_2 or WF server S_3. In addition, it

will be not possible to determine which WF server controls a parallel branch, without further effort, if variable server assignments are used. In Figure 6, for example, the WF server assignment of activity e refers to the actor of activity c, which is not known by WF server s_4. – In the following, we restrict our considerations to Alternative 3.

Determining the Set of Active Servers of a Workflow Instance

As explained above, generally, a WF server is not always able to determine from its local state information which other WF servers are currently executing activities of a specific WF instance. And it is no good idea to use a broadcast call to search for these WF servers, as this would result in exactly the same drawbacks as described above for the naive solution. We, therefore, require an approach for explicitly managing the active WF servers of a WF instance. The administration of these WF servers, however, should not be carried out by a fixed (and therefore central) WF server since this might lead to bottlenecks, thus negatively impacting the availability of the whole WfMS. For this reason, in ADEPT, the set of active WF servers (*ActiveServers*) is managed by a *ServerManager* specific to the WF instance. For this purpose, for example, the start server of the WF instance can be used as *ServerManager*. Normally, this WF server varies for each of the WF instances (even if they are of the same WF type), thus avoiding bottlenecks.[3]

The start WF server can be easily determined from the local execution history by any WF server involved in the control of the WF instance. In the following we show how the set of active servers of a specific WF instance is managed by the *ServerManager*, how it can be determined, and how ad-hoc changes can be synchronized.

Managing Active WF Servers of a WF Instance

As aforementioned, for the ad-hoc change of a WF instance we require the set *ActiveServers*, which comprises all WF servers currently involved in the control of the WF instance. This set, which may be changed due to migrations, is explicitly managed by the *ServerManager*. Thereby, the following two rules have to be considered:

1. Multiple migrations of the same WF instance must not overlap arbitrarily, since this would lead to inconsistencies when changing the set of active WF servers.
2. For a given WF instance the set *ActiveServers* must not change due to migrations during the execution of an ad-hoc change. Otherwise, wrong WF servers would be involved in the ad-hoc change or necessary WF servers would be left out.

As we will see in the following, we prevent these two cases by the use of several locks.[4] In the following, we describe the algorithms necessary to satisfy these

[3] Using this policy there may be scenarios where the same WF server would be always used as all WF instances in the WfMS are created on the same WF server. (An excellent example is the server that manages the terminals used by the tellers in a bank.) In this case, the *ServerManager* should be selected arbitrarily when a WF instance is generated.

[4] A robust behavior of the distributed WfMS could also be achieved by performing each ad-hoc change and each migration (incl. the adaptation of the set *ActiveServers*) within a distributed transaction (with 2-phase-commit). But this approach would be very restrictive since during the execution of such an operation, "normal WF execution" would be prevented. That means, while performing a migration, the whole WF instance would be locked and, therefore, even the execution of activities actually not

requirements. *Algorithm 1* shows the way migrations are performed in ADEPT. It interacts with *Algorithm 2* by calling procedure *UpdateActiveServers* (remotely), which is defined by this algorithm. This procedure manages the set of active WF servers currently involved in the WF instance; i.e., it updates this set consistently in case of WF server changes.

Algorithm 1 illustrates how a migration is carried out in our approach:

Algorithm 1 (Performing a Migration)

```
input
    Inst: ID of the WF instance to be migrated
    SourceServer: source server of the migration (it performs Algorithm 1)
    TargetServer: target server of the migration
begin
    // determine the ServerManager for this WF instance from its execution history
    ServerManager = StartServer(Inst);
    // request a non-exclusive lock and an exclusive short-term lock from the ServerManager
    RequestSharedLock(Inst) → ServerManager;[5]
    RequestShortTermLock(Inst) → ServerManager;
    // change the set of active servers (cf. Algorithm 2)
    if LastBranch(Inst) then
        // the migration is performed for the last execution branch of the WF instance, that
        // is active at the SourceServer
        UpdateActiveServers(Inst, SourceServer, LogOff, TargetServer) → ServerManager;
    else  // another execution path is active at SourceServer
        UpdateActiveServers(Inst, SourceServer, Stay, TargetServer)  → ServerManager;

    // perform the actual migration and release the non-exclusive lock
    MigrateWorkflowInstance(Inst) → TargetServer;
    ReleaseSharedLock(Inst) → ServerManager;
end.
```

Algorithm 1 is initiated and executed by a source WF server that hands over control to a target WF server. First, the *SourceServer* requests a non-exclusive lock from the *ServerManager*, which prevents that the migration is performed during an ad-hoc change.[6] Then an exclusive, short-term lock is requested. This lock ensures that the *ActiveServers* set of a given WF instance is not changed simultaneously by several migrations within parallel branches. (Both lock requests may be incorporated into a single call to save a communication cycle.) The *SourceServer* reports the change of the *ActiveServers* set to the *ServerManager*, specifying whether it remains active for the concerned WF instance (*Stay*), or whether it is not involved any longer (*LogOff*). If, for example, in Figure 6 the migration $M_{b,c}$ is executed before $M_{f,g}$, the option *Stay* will be used for the migration $M_{b,c}$ since WF server S_1 remains active for this WF instance. Thus, the option *LogOff* will be used for the subsequent migration $M_{f,g}$ as it ends the last branch controlled by S_1. The (exclusive) short-term lock prevents that these two migrations may be executed simultaneously. This ensures that it is always clear whether or not a WF server remains active for a WF instance when a migration

concerned would not be possible. Such a restrictive approach is not acceptable for any WfMS. However, it is not required in our approach and we realize a higher degree of parallel execution while achieving the same robustness.

[5] *p() → s* means that procedure *p* is called and then executed by server *s*.

[6] For details see Algorithm 3. The lock does not prevent several migrations of one and the same WF instance from being performed simultaneously.

has completed. Next, the WF instance data (e.g., the current state of the WF instance) is transmitted to the target WF server of the migration. Since this is done after the exclusive short-term lock has been released (by *UpdateActiveServers*), several migrations of the same WF instance may be executed simultaneously. The algorithm ends with the release of the non-exclusive lock.

Algorithm 2 is used by the *ServerManager* to manage the WF servers currently involved in controlling a given WF instance. To fulfill this task, the *ServerManager* also has to manage the locks mentioned above. If the procedure *UpdateActiveServers* is called with the option *LogOff*, the source WF server of the migration is deleted from the set *ActiveServers(Inst)* (i.e., the set of active WF servers with respect to the given WF instance). The reason for this is that this WF server is no longer involved in controlling this WF instance. The target WF server for the migration, however, is always inserted into this set independently of whether it is already contained or not because this operation is idempotent.

The short-term lock requested by Algorithm 1 before the invocation of *Update-ActiveServers* prevents Algorithm 2 from being run in parallel more than once for a given WF instance. This helps to avoid an error due to overlapping changes of the set *ActiveServers(Inst)*. When this set has been adapted, the short-term lock is released.

Algorithm 2 (*UpdateActiveServers*: Managing the active WF Servers)

```
input
    Inst: ID of the affected WF instance
    SourceServer: source server of the migration
    Option: indicates whether source server is further involved in the WF instance
            (Stay) or not (LogOff)
    TargetServer: target server of the migration
begin
    // update the set of active  WF servers of the WF instance Inst
    if Option = LogOff  then
        ActiveServers(Inst) = ActiveServers(Inst)  − {SourceServer};

    ActiveServers(Inst) = ActiveServers(Inst) ∪ {TargetServer};
    ReleaseShortTermLock(Inst);  // release the short-term lock
end.
```

Performing Ad-hoc Changes

While the previous subsection has described how the *ServerManager* handles the set of currently active WF servers for a particular WF instance, we now show how this set is utilized when ad-hoc changes are performed.

First of all, if no parallel branches are currently executed, trivially, the set of active WF servers contains exactly one element, namely the current WF server. This case may be easily detected by making use of the state and structure information (locally) available at the current WF server. The same applies to the special case that currently all parallel branches are controlled by the same WF server. In both cases, the method described in the following is not needed and therefore not applied. Instead, the WF server currently controlling the WF instance performs the ad-hoc change without consulting any other WF server. Consequently, this WF server need also not communicate with the *ServerManager*. For this special case, therefore, no additional synchronization effort occurs (when compared to the central case).

We now consider the case that parallel branches exist; i.e., an ad-hoc change of the WF instance may have to be synchronized between multiple WF servers. The WF server which coordinates the ad-hoc change then requests the set *ActiveServers* from the *ServerManager*. When performing the ad-hoc change, it is essential that this set is not changed due to concurrent migrations. Otherwise, wrong WF servers would be involved in the change procedure. In addition, it is vital that the WF execution schema of the WF instance is not restructured due to concurrent modifications, since this may result in an incorrect schema.

To prevent either of these errors we introduce ***Algorithm 3***. It requests an exclusive lock from the *ServerManager* to avoid the aforementioned conflicts. This lock corresponds to a write lock (Gray, 1993) in a database system and is incompatible with read locks (*RequestSharedLock* in Algorithm 1) and other write locks of the same WF instance. Thus, it prevents that migrations are performed simultaneously to an ad-hoc change of the WF instance.

Algorithm 3 (Performing an Ad-hoc Change)

```
input
    Inst: ID of the WF instance to be modified
    Modification: specification of the ad-hoc change
begin
    // calculate the ServerManager for this WF instance
    ServerManager = StartServer(Inst);
    // request an exclusive lock from the ServerManager and calculate the set of active WF servers
    RequestExclusiveLock(Inst) → ServerManager;
    ActiveServers = GetActiveServers(Inst) → ServerManager;
    // request a lock from all servers, calculate the current WF state, and perform
    // the change (if possible)
    for each Server s ∈ ActiveServers do
        RequestStateLock(Inst) → s;
    GlobalState = GetLocalState(Inst);
    for each Server s ∈ ActiveServers do
        LocalState = GetLocalState(Inst) → s;
        GlobalState = GlobalState ∪ LocalState;
    if DynamicModificationPossible(Inst, GlobalState, Modification) then
        for each Server s ∈ ActiveServers do
            PerformDynamicModification(Inst, GlobalState, Modification) → s;
    // release all locks
    for each Server s ∈ ActiveServers do
        ReleaseStateLock(Inst) → s;
    ReleaseExclusiveLock(Inst) → ServerManager;
end.
```

As soon as the lock has been granted in *Algorithm 3*, a query is sent to acquire the set of active WF servers of this WF instance.[7] Then a lock is requested at all WF servers belonging to the set *ActiveServers* in order to prevent local changes to the state of the WF instance. Any activities already started, however, may be finished normally since this does not affect the applicability of an ad-hoc change. Next the (locked) state information is retrieved from all active WF servers. Remember that the resulting global and current state of the WF instance is required to check whether the ad-hoc modification to be performed is permissible or not. In Figure 6, for example, WF server s_4, which is currently controlling activity g and which wants to insert activity

[7] This query may be combined with the lock request into a single call to save one communication cycle.

x after activity g and before activity d, normally does not know the current state of activity d (from the parallel branch). Yet the ad-hoc change will be permissible only if activity d has not been started at the time the change is initiated (Reichert, 1998a). If this is the case, the ad-hoc change is performed at all active WF servers of the WF instance (*PerformDynamicModification*). Afterwards, the locks are released and any blocked migrations or modification procedures may then be carried out.

Illustrating Example

How migrations and ad-hoc changes work together is explained by means of an example. Figure 7a shows a WF instance currently controlled by exactly one WF server, i.e. WF server s_1. Figure 7b shows the same WF instance after it migrated to a second WF server s_2. In Figure 7c execution was continued. One can also see that each of the two WF servers must not always possess complete information about the global state of the WF instance.

Assume now that an ad-hoc change shall be performed, which is coordinated by WF server s_1. Afterwards, both WF servers shall possess the current schema of the WF instance to correctly proceed with the flow of control. With respect to the (complete) current state of the WF instance, it is sufficient that it is known by the coordinator s_1 (since only this WF server has to decide on the applicability of the desired change). The other WF server only carries out the change (as specified by WF server s_1).

DISTRIBUTED EXECUTION OF A MODIFIED WORKFLOW INSTANCE

If a migration of a WF instance has to be performed its current state has to be transmitted to the target WF server. In ADEPT, this is done by transmitting the relevant parts of the execution history of the WF instance together with the values of WF relevant data (i.e., data elements) (Bauer, 2001). If an ad-hoc change was previously performed, the target WF server of a migration also needs to know the modified execution schema of the WF instance in order to be able to control the WF instance correctly afterwards. In the approach introduced in the previous section, only the active WF servers of the WF instance to be modified have been involved in the ad-hoc change. Consequently, the WF servers controlling subsequent activities still have to be informed about the conducted change. In our approach, the necessary information is transmitted upon migration of the WF instance to the WF servers in question. Since migrations are rather frequently performed in distributed WfMS, this communication needs to be performed efficiently. We first introduce a method that meets this requirement to a satisfactory degree. Then we present an enhancement that additionally precludes redundant data transfer.

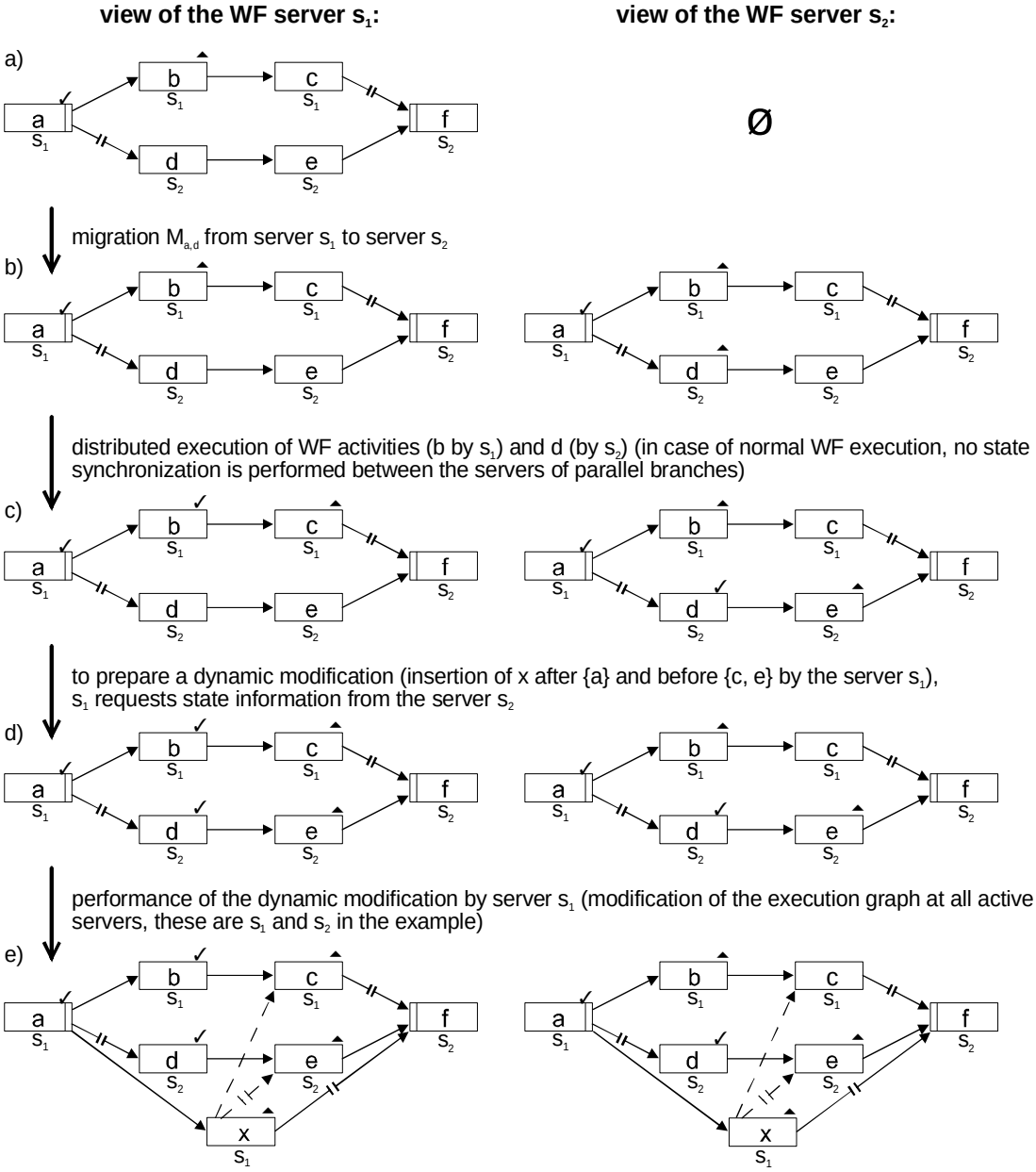


Figure 7. Effects of migrations and ad-hoc changes on the (distributed) execution schema of a WF instance (local view of the WF servers)

Efficient Transmission of Data about Ad-hoc Changes

In the following, we examine how a changed WF execution schema can be communicated to the target WF server of a migration. The key objective of this investigation is the development of an efficient technique that reduces communication-related costs as best as possible. Obviously, the simplest way to communicate the current execution schema of the respective WF instance to the migration target server is to transmit this schema in whole. Yet this technique burdens the communication system unnecessarily because the related WF graph of this WF schema may comprise a large number of nodes and edges. This results in an enormous amount of data to be transferred – an inefficient and cost-intensive approach.

Apart from this, the entire execution schema does not need to be transmitted to the migration target server as the related WF template has been already located there. (Note that a WF template is being deployed to all relevant WF servers before any WF instance may be created from it.) In fact, in most cases the current WF schema of the WF instance is almost identical to the WF schema associated with the WF template. Thus it is more efficient to transfer solely the relatively small amount of data which specifies the change operations applied to the WF instance; i.e., to use the change history for this purpose. In the ADEPT approach, the migration target server needs this history anyway (Reichert, 1998a; Rinderle, 2006a), so that its transmission does not lead to additional efforts. When the base operations recorded in the change history are applied to the original WF schema of the WF template, the result is the current WF schema of the given WF instance. This simple technique significantly reduces communication efforts. In addition, as typically only very few changes are performed on any individual WF instance, computation time is kept to a minimum.

Enhancing the Method used to Transmit Change Histories

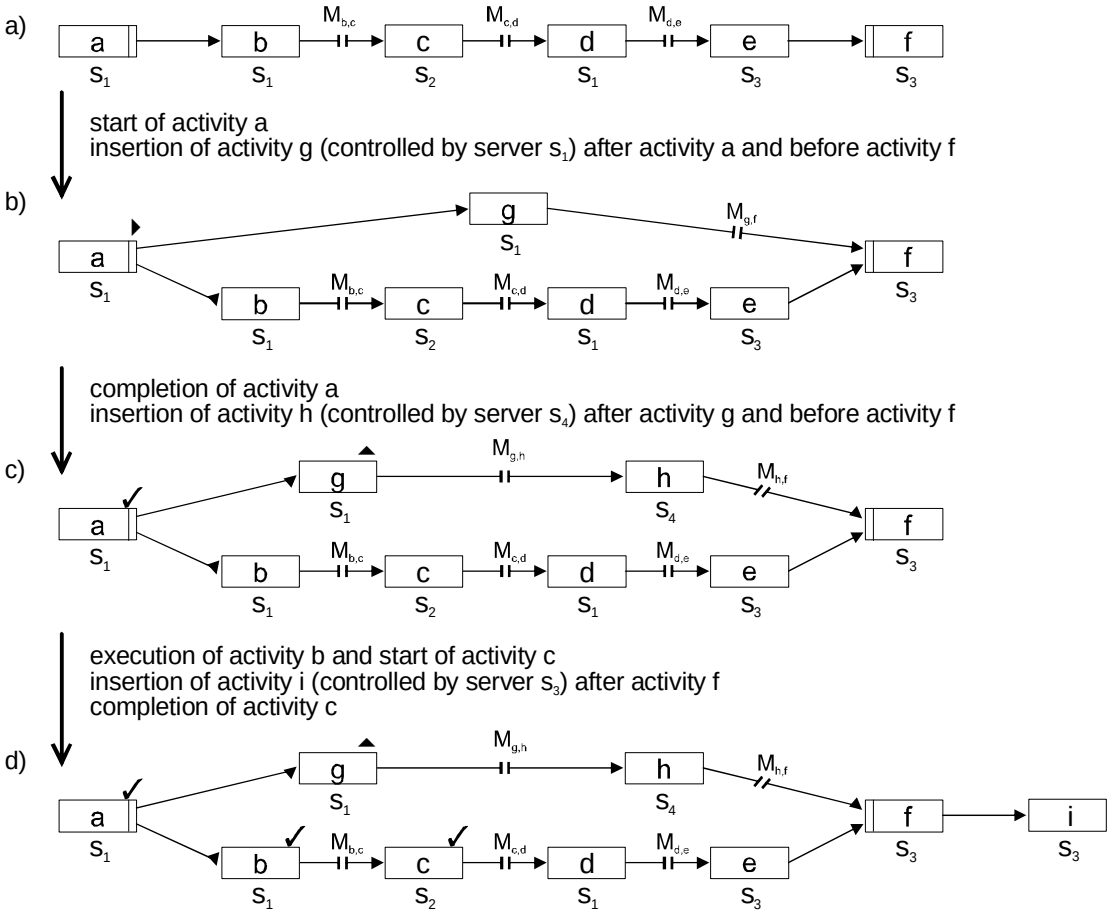
Generally, one and the same WF server can be involved more than once in the execution of a WF instance – especially in conjunction with loop backs. In our example from Figure 8, for instance, WF server s_1 hands over control to WF server s_2 after completion of activity b, but will receive control again later on in the flow to execute activity d. Since each WF server stores the change history until being informed that the given WF instance has been completed, such a WF server s already knows the history entries of the changes it has performed itself. In addition, s knows any changes that had been effected by other WF servers before s handed over the control of the WF instance to another WF server for the last time. Hence the data related to this part of the change history need not be transmitted to the WF server. This further reduces the amount of data required for the migration of the "current execution schema".

Transmitting Change History Entries

An obvious solution for avoiding redundant transfer of change history entries is as follows: The migration source server determines from the existing execution history exactly which changes the target WF server does already know. The related entries are then simply not transmitted when migrating the WF instance. In the example given in Figure 8, WF server s_2 can determine, upon ending activity c, that the migration target server s_1 does already know Changes 1 and 2. In the execution history (cf. Figure 8e), references to these changes (*DynModif(1)* and *DynModif(2)*) have been recorded before entry *End(b, s_1, ...)* (which was logged when completing activity b). As this activity was controlled by WF server s_1, this WF server does already know the Changes 1 and 2. Thus, for the migration $M_{c,d}$, only the change history entry corresponding to Change 3 needs to be transmitted. The transmitted part of the change history is concatenated with the part already being present at the target server before this WF server creates the new execution schema and proceeds with the flow of control.

In some cases, however, redundant transfer of change history data cannot be avoided with this approach. As example take migrations $M_{d,e}$ and $M_{h,f}$ to WF server s_3. For

both migrations, using the above approach, all entries corresponding to Changes~1, 2, and 3 must be transmitted since WF server s_3 was not involved in the execution of the WF instance thus far. The problem is that migration source servers s_1 and s_4 are unable, from their locally available history data, to derive whether the other migration from the parallel branch has already been effected or not. Therefore, the entire change history has to be transmitted. Yet with the more advanced approach set out in the following, we can avoid such redundant data transfer.



e) Start(a, s_1, ...), DynModif(1), End(a, s_1, ...), DynModif(2), Start(b, s_1, ...), End(b, s_1, ...), Start(c, s_2, ...), DynModif(3), End(c, s_2, ...)

Figure 8. (a-d) WF instance and (e) execution history of WF server s_2 after completion of activity c. – In case of distributed WF control, with each entry the execution history records the server being responsible for the control of the corresponding activity.

Requesting Change History Entries

To avoid redundant data transmissions, we introduce a more sophisticated method. With this method, the necessary change history entries are explicitly requested by the migration target server. When a migration takes place, the target WF server informs the source WF server about the history entries it already knows. The source WF server then only transmits those change history entries of the WF instance yet missing on the side of the target server. In ADEPT, a similar method has been used for transmitting execution histories; i.e., necessary data is provided on basis of a request from the migration target server (Bauer, 2001). Here, no additional effort is expended for

communication, since both the request for and the transmission of change history entries may be carried out within same communication cycle.

With the described method, requesting the missing part of a change history is efficient and easy to implement in our approach. If the migration target server was previously involved in the control of the WF instance, it would already possess all entries of the change history up to a certain point (i.e., it knows all ad-hoc changes that had been applied to the respective WF instance before this server handed over control the last time). But from this point on, it does not know any further entries. It is thus sufficient to transfer the ID of the last known entry to the migration source server to specify the required change history entries. The source WF server then transmits all change history entries made after this point.

The method described above is implemented by means of ***Algorithm 4***, which is executed by the migration source server as part of the *MigrateWorkflowInstance* procedure (cf. Algorithm 1). This procedure also effects transmission of the execution history and of WF relevant data. *Algorithm 4* triggers the transmission of the change history by requesting the ID of the last known change history entry from the target WF server. If no change history for the given WF instance is known at the target WF server it will return `NULL`. In this case, the entire change history is relevant for the migration and is therefore transmitted to the target WF server. Otherwise, the target WF server requires only that part of the change history, which follows the specified entry. This part is copied into the history *RelevantChangeHistory* and transmitted to the target WF server. This data may be transmitted together with the mentioned WF execution data to save a communication cycle.

Algorithm 4 (Transmission of Change History Data)

```
input
    Inst: ID of the WF instance to be changed
    TargetServer: server, which receives the change history
begin
    // start the transmission of the change history by asking for the ID of the last known entry
    LastEntry = GetLastEntry(Inst) → TargetServer;
    // calculate the relevant part of the change history
    if LastEntry = NULL then  // change history totally unknown at the target WF server
        Relevant = True;
    else    // all entries until LastEntry (incl.) are known by the target server
        Relevant = False;
    // initialize the position counters for the original and the new change history
     i = 1;  j = 1;
    // read the whole change history of WF instance Inst
    while ChangeHistory(Inst)[i] ≠ EOF} do
        if Relevant = True then    // put the entry in the result (if necessary)
            RelevantChangeHistory[j] = ChangeHistory(Inst)[i];
            j =  j + 1;
        // check whether the end of that part of the change history, that is known by the
        // target WF server, is reached
        if EntryID(ChangeHistory(Inst)[i]) = LastEntry then
            Relevant = True;
        i = i + 1;
    // perform the transmission of the change history
    TransmitChange(Inst, RelevantChangeHistory) → TargetServer;
end.
```

Algorithm 4 is illustrated by means of the example given in Figure 8: Concerning the migration $M_{c,d}$ the target WF server S_1 already knows the ad-hoc changes 1 and 2. Thus it responds to the request of the source server with *LastEntry = 2*. The migration source server then ignores the change history entries for changes 1 and 2, transmitting only the entry for change 3 to target WF server S_1. This result is identical to the one achieved in the approach for transmitting change history entries.

For the migrations $M_{h,f}$ and $M_{d,e}$, without loss of generality, it is assumed that $M_{h,f}$ is executed before $M_{d,e}$.[8] Since there has been no change history of this WF instance located on WF server S_3 yet, the target WF server of migration $M_{h,f}$ returns *LastEntry = NULL*. Therefore, the entire change history is transmitted to S_3. In the subsequent migration $M_{d,e}$, the target WF server S_3 then already knows change history entries 1 – 3, so that *LastEntry = 3* will be returned in response to the source server query. (When the *while* loop in Alg. 4 is run, variable *Relevant* is not set to *True* until history entries 1 – 3 have been processed. Since there exist no further entries in the change history, *RelevantModificationHistory* remains empty with the result that no change history entries have to be transmitted.) Finally, the problem of redundant data transfer, as described at the beginning of this section, is thus avoided here.

To sum up, with the described approach not only ad-hoc modifications can be performed efficiently in a distributed WfMS, but transmission costs for migrating changed WF instances can be kept low as well.

PROOF-OF-CONCEPT PROTOTYPE

All methods presented in this chapter have been implemented in a powerful proof-of-concept prototype. It demonstrates feasibility of ad-hoc changes in a distributed WfMS and shows how the developed concepts work in conjunction with each other.

Buildtime Components

Our proof-of-concept prototype supports the WF designer by powerful tools. They support the definition of WF templates, the modeling of organizational entities and their relationships, the specification of access control constraints (e.g., authorizations concerning WF changes; Weber, 2005b), and the plug-in of application services. All relevant information is stored in a repository. In addition, XML-based descriptions of the defined models can be created; e.g., to export them to other tools or to deploy them to the WF servers of the distributed WfMS.

For WF modeling we offer a syntax-driven, graphical WF editor. A sample screen is depicted in Figure 9. It shows a clinical workflow as modeled in ADEPT. The upper part of this screen shows the control flow of this workflow, whereas the lower part displays the input parameters of the currently selected activity *calculate dose* (as well as the mapping of these parameters to data elements). Additional information about

[8] A lock at the target WF server prevents the migrations from being carried out concurrently in an uncoordinated manner. This ensures that migrations for one and the same WF instance are serialized; i.e., the lock is maintained from start of migration, while change history entries (and other WF-related data (Bauer, 2001)) are acquired and transmitted, until the entries have finally been integrated into the change history at the target WF server. This lock prevents history entries from being requested redundantly due to the request being based on obsolete local information.

the selected activity is shown on the right-hand side. Further down, a pacemaker box is displayed, which helps the WF designer to navigate through larger models. We explain the WF model from Figure 9 in more detail, since we refer to it in the following. This model describes the medication of a patient during a treatment cycle in a hospital. The workflow starts with the patient's admission to a ward (by a *ward nurse*). It then proceeds with activities *instruct patient* (by *ward physician*) and *collect patient data* (by *ward nurse*). Afterwards, the flow splits into two parallel branches which may be executed concurrently. The upper branch comprises the activities of a medical examination performed in another department (*perform examination* and *write report* both with user role *radiologist*), whereas the lower branch defines preparatory steps performed at the ward side (e.g., *calculate dose*, *produce drug*). These two branches contain some other activities (*read report*, *validate dose*) not displayed in Figure 9. When both branches are completed, they are joined and the produced drug is administered to the patient, some aftercare is provided, and the patient is discharged (also not displayed in Figure 9).

Our prototype supports the WF designer in calculating optimal WF server assignments for the respective WF activities; i.e., in partitioning the WF schema such that overall communication costs become minimal at runtime. For this purpose, we have implemented advanced algorithms which make use of the information from the organizational model (i.e., roles as well as locations of actors). Concerning our example from Figure 9, respective WF instances are controlled by WF servers s_1 and s_2. The calculated WF server assignments are displayed below the activity nodes. Accordingly, activities *perform examination* and *write report* are controlled by WF server s_2, whereas all other activities are carried out by WF server s_1.

Furthermore, the developed WF editor supports the designer in modeling error-free WF templates (e.g., by excluding deadlocks and by ensuring data flow correctness) – we denote this capability as *correctness by construction*. To achieve it, both on-the-fly checks during editing and complete model checks initiated by the designer at any point in time are possible. In any case, a new WF template may only be released, if all correctness and consistency checks are successfully passed. Note that this is fundamental for the support of ad-hoc changes as well. An adaptive WfMS will only be able to guarantee consistency if a WF instance is consistent before a change as well. This, in turn, is crucial for the WfMS to guarantee a reliable and robust execution behavior of the distributed WF instances.

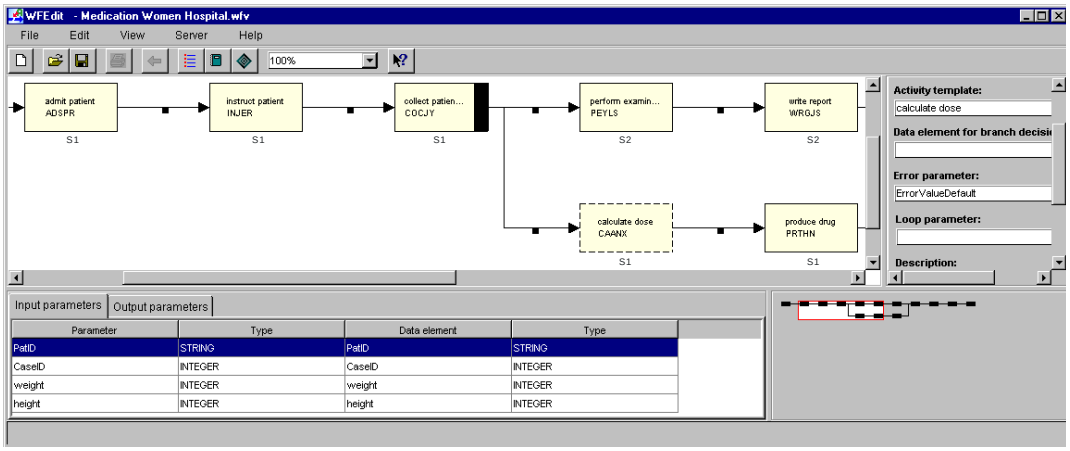


Figure 9. Workflow Editor

A new release of a WF template can be introduced to the distributed WfMS by deploying it to relevant WF servers. For this, an XML-based description is sent to the WF servers and imported into their run-time databases. – We omit descriptions of other build-time components since they are not relevant in the context of this paper (Reichert, 2003c).

Runtime Components

Our proof-of-concept prototype comprises run-time clients for end users, process administrators, and system administrators. They provide support for configuring the distributed WfMS, for managing WF instances, for handling user worklists, for defining ad-hoc WF changes, and for monitoring WF instance execution.

To monitor the progress of WF instances in ADEPT and to demonstrate the effects of ad-hoc changes, we offer a monitoring component. It enables authorized users (e.g., process administrators) to visualize the execution schema of a WF instance together with the information related to that WF instance. A sample screen is depicted in Figure 10. It shows the execution schema of a WF instance which was created from the WF template depicted in Figure 9. Activities admit patient, instruct patient and collect patient data are completed (indicated by symbol ✓), whereas activity calculate dose is currently activated (expressed by symbol □). The upper part of Figure 10 displays the data elements read and written by the currently selected activity (calculate dose in this example) as well as detailed information about the activity (e.g., state, actor assignment, execution mode, server assignment, earliest / latest starting times, etc.). All relevant information is managed by WF server S_1 which controls activity calculate dose. We provides a powerful *application programming interface* for accessing respective information.

Actually, the depicted monitoring client only shows the execution schema from the viewpoint of WF server S_1 (to which this client is connected). However, WF server S_1 does not know how far execution has proceeded in the upper branch of the parallel branching (currently controlled by WF server S_2). For example, WF server S_1 does not know whether activity perform examination has been activated, started, or completed yet.

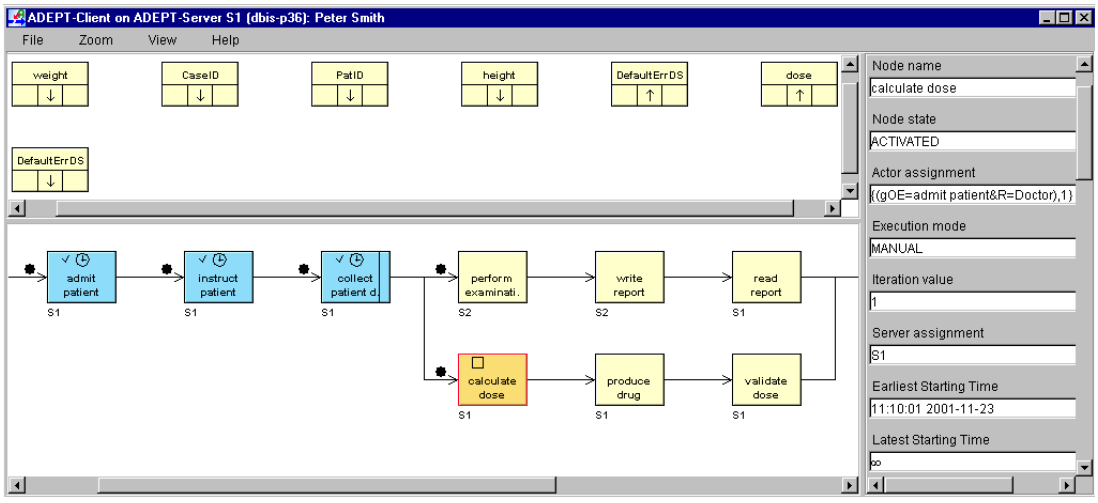


Figure 10. Monitoring client (before applying an ad-hoc change to the depicted WF instance)

How can an ad-hoc change be realized in the given scenario? End users must be able to define such change at a high level of abstraction; i.e., without need to be familiar with the WF editor or to have knowledge about distributed execution of the WF instance. For this purpose we offer easy-to-use runtime clients to the actors.[9]

We now come back to our WF instance from Figure 10. Assume that an authorized user (connected to WF server S_1) wants to insert activity perform allergy test after activity instruct patient and before activities write report and produce drug; i.e., the user wants the allergy test to be started after instruction of the patient and to be completed before a report is written and the drug is produced. If this change is applied to the WF instance from Figure 10, the execution schema from Figure 11 will result. Here, node n1 represents an AND-split which resulted from the transformation of the change into respective schema adaptations (Reichert, 1998a; Dadam, 1998). Also note that state information from WF server S_2 had to be retrieved and the techniques presented in the previous sections were applied.

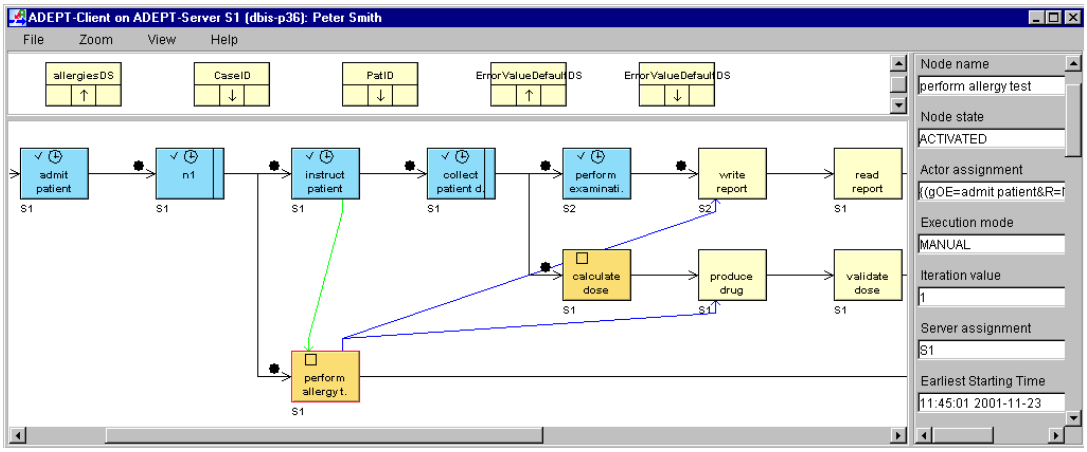


Figure 11. Monitoring client (after applying the ad-hoc change to the depicted WF instance)

DISCUSSION

In literature, we can find a number of approaches addressing issues related to scalability and distributed WF execution. Besides centralized WfMS, which include most commercial systems (e.g. Staffware (Staffware, 2003)), several distributed WfMS consisting of multiple WF servers exist. Some of them assign a WF instance (as a whole) always to the same WF server. Examples include Exotica/Cluster (Alonso, 1994) and MOBILE (Jablonski, 1997); the latter approach was extended in (Schuster, 1999). Comparable to our techniques, the approaches provided by MENTOR (Muth, 1998) and WIDE (Casati, 1996) select the WF server for a WF activity next to its potential actors. CodAlf, BPAFrame (Schill, 1996), and METEOR$_2$ (Sheth, 1997), in turn, allocate the WF server for a WF activity on that node where its corresponding application service is located. Furthermore, completely distributed WfMS, like Exotica/FMQM (Alonso, 1995) and INCAs (Barbará, 1996}, use the

[9] To enable application developers to implement customized runtime components, we provide a powerful application programming interface (API) to them. Its functionality goes far beyond the APIs of existing WfMS. For example, our API provides powerful change operations, which hide as much of the complexity of an ad-hoc change as possible from users.

machines of the actors as WF servers. Finally, there are approaches for distributed WF management, which do not have a special strategy for distributing the activities to the WF servers; e.g., EVE (Geppert, 1998), METUFlow (Dogac, 1997), MOKASSIN (Gronemann, 1999), $WASA_2$ (Weske, 1998; Weske, 1999), and the Petri-net based approach presented in (Guth, 1998).

Similarly, many groups deal with issues related to ad-hoc WF changes. They focus on different issues arising in this context. Like ADEPT (Reichert, 1998a), Chautauqua (Ellis, 1997), $WASA_2$ (Weske, 1998), and WF nets (van der Aalst, 2001a+b) deal with issues related to the correctness and consistency of modified WF instances. CBRFlow (Weber, 2004), ProCycle (Rinderle, 2005; Weber, 2005a; Weber, 2006b), and CAKE2 (Minor, 2007), for example, additionally apply knowledge-based techniques (e.g., case-based reasoning) to increase WfMS flexibility and to foster the reuse of ad-hoc changes. The approaches described in (van der Aalst, 2001b) uses generic WF models to deal with dynamic WF changes. In this context, a generic WF model describes a family of WF models (i.e., model variants) of the same WF type. Consequently, an (ad-hoc) change is handled by migrating a WF instance between different members of the same process family. This is supported by defining a minimal representative for each process family and by specifying rules for transferring a variant to the minimal representative (and vice versa). An approach based on inheritance, which uses generic inheritance-preserving transformation and transfer rules, is suggested by (van der Aalst, 2002). With this approach, certain errors in connection with changes can be avoided by choosing appropriate inheritance notions. Finally, there are several approaches aiming at the support of WF schema evolution and the propagation of the resulting schema changes to already running WF instances (if compliant to the new scheme). Corresponding work has been done in MOKASSIN (Joeris, 1998), WIDE (Casati, 1998), TRAM (Kradolfer, 1999), ADEPT2 (Rinderle, 2004a-c; Rinderle-Ma, 2008a), and $WASA_2$ (Weske, 1998).

There are only few projects which allow for ad-hoc changes as well as distributed WF control. In particular, how these two fundamental features of a large-scale WfMS impact each other has not yet been investigated in detail. The major objective of the aforementioned approaches was not to develop a scalable and flexible WfMS which is efficient with regard to communication costs. This has been systematically investigated in this chapter.

There are few approaches which address both WF changes and distributed WF execution. WIDE allows WF schema changes and their propagation to running WF instances (Casati, 1998). In addition, control of WF instances can be distributed (Casati, 1996). Thereby, the set of potential actors of an activity determines the WF server which has to control this activity. In MOKASSIN (Gronemann, 1999; Joeris, 1998} and $WASA_2$ (Weske, 1998+1999), distributed WF execution is realized through an underlying CORBA infrastructure. Both approaches do not discuss the criteria used to determine a concrete distribution of the WF activities; i.e., the question which WF server has to control a specific activity remains open. Here, changes may be applied at both the WF schema and the WF instance level.

INCAs (Barbará, 1996) uses rules for WF instance coordination. WF control is distributed with a given WF instance being controlled by that processing station that belongs to the actor of the current activity. The mentioned rules are used to calculate the processing station of the subsequent activity and, thereby, the actor of that activity. With this approach, it becomes possible to modify the rules, what results in

an ad-hoc change of the WF instance behavior. As opposed to our approach, none of these works explicitly addresses how ad-hoc changes and distributed WF execution interact. The approach proposed in (Cichocki, 2000) enables some kind of flexibility in distributed WfMS as well, especially in the context of virtual enterprises. However, it does not allow to adapt the structure of in-progress WF instances. Instead, the activities of a WF template represent placeholders for which the concrete implementations are selected at run-time – a similar approach is provided by pockets of flexibility (Sadiq, 2001; Sadiq, 2005). Finally, DYCHOR allows for structural changes of process choreographies, but without taking state information into account (Rinderle, 2006b).

There are several approaches for distributed WF management where a WF instance is controlled by one and the same WF server over its entire lifetime; e.g., Exotica (Alonso, 1994) and MOBILE (Jablonski, 1997). The latter approach was extended in (Schuster, 1999) such way that a sub-process may be controlled by a different WF server to be determined at run-time. Though migrations are not performed, different WF instances may be controlled by different WF servers. Furthermore, since a central control component (i.e., WF engine) exists for each WF instance in these approaches, ad-hoc changes may be performed just as in a central WfMS. Yet there is a drawback with respect to communication costs (Bauer, 1999) since the distribution model does not allow to select the most favorable WF server for the individual activities. When developing our approach, we therefore did not follow such an approach since the additional costs incurred in standard WF execution are higher than the savings generated due to the (relatively seldom performed) ad-hoc changes.

SUMMARY AND OUTLOOK

Both distributed WF execution and ad-hoc WF changes are essential features of any WfMS in order to enable flexible process-aware information systems. However, each of these aspects is closely linked with a number of requirements and objectives that are, to some extent, opposing. Typically, the central control instance required for ad-hoc changes impacts the efficiency of (distributed) WF execution. For these reasons we cannot afford to consider these two fundamental aspects separately. In this book chapter, an investigation of exactly how these two features interact has been presented. Our results have shown that are, in fact, compatible. We have realized ad-hoc changes in a distributed WfMS efficiently. Our approach also allows for the efficient distributed control of changed WF instances. The described techniques make use of the fact that only a parts of the relatively small change history need to be transmitted when transferring a modified WF execution schema to another WF server. This is vital as migrations are frequently performed operations. As demonstrated with our proof-of-concept prototype, our approach succeeds in seamlessly integrating both distributed WF execution and ad-hoc changes into a single system.

There is room for further optimization regarding the selection of the WF servers that need to be synchronized in the context of an ad-hoc change. If such a change affects only a particular region of the WF schema, it could be performed by only those active WF servers controlling that region of the WF instance. This would reduce synchronization and communication efforts. In the extreme case, if only a single branch of a parallel branching has to be changed, only a single server must perform the change. However, activities belonging to parallel branches may be impacted by the change performed (e.g. due to dependencies in the data flow), thus necessitating

synchronization of the respective WF servers in these cases. Our work has shown that the opportunity to deploy such an enhancement is fairly rare so that a significant improvement in the behavior of the system cannot be expected.

Generally, non-trivial interdependencies exist among the different features of a WfMS, which should be carefully analyzed and understood. One cannot implement such WfMS by adding one balcony to the other to deal with situation-specific problems. Instead a proper framework is needed which allows to argue about WF correctness and which covers all possible scenarios. The ADEPT1 project (Dadam, 1998) has reflected this way of thinking from the very beginning. In ADEPT2 (Reichert, 2005), we have extended respective research activities to other aspects as well, e.g., concerning the mining and evolution of access control constraints (Ly, 2005; Rinderle-Ma, 2007+2008c) or different techniques for learning from ad-hoc changes (Rinderle, 2005; Günther, 2006; Li, 2008). Recently, we have started our research on WF change patterns (Weber, 2007+ 2008a+2009; Rinderle-Ma, 2008b), WF refactoring techniques (Weber, 2008b), and data-driven WF coordination and adaptation (Müller, 2007+2008).

REFERENCES

van der Aalst, W.M.P., & ter Hofstede, A. (2000). Verification of workflow task structures: a Petri-net-based approach. *Information Systems*, 25(1), 43–69.

van der Aalst, W.M.P. (2001 a). Exterminating the dynamic change bug: a concrete approach to support workflow change. *Information Systems Frontiers*, 3(3), 297–317

van der Aalst, W.M.P. (2001 b). How to handle dynamic change and capture management information: an approach based on generic workflow models. *Int. Journal of Computer Systems, Science, and Engineering*, 16(5), 295–318.

van der Aalst, W.M.P., & Basten, T. (2002). Inheritance of workflows: an approach to tackling problems related to change. *Theoretical Computer Science*, 270(1-2), 125-203.

Adams, M., ter Hofstede, A., Edmond, D., & van der Aalst, W.M.P. (2006). Worklets: a service-oriented implementation of dynamic flexibility in workflows, In: *Proc. Coopis'06* (pp. 291-308).

Alonso, G., Kamath, M., Agrawal, D., El Abbadi, A., Günthör, R., & Mohan, C. (1994). Failure handling in large scale workflow management systems. *TR RJ9913, IBM Almaden Research Center*.

Alonso, G., Mohan, C., Günthör, R., Agrawal, D., El Abbadi, A., & Kamath, M. (1995). Exotica/ FMQM: persistent message-based architecture for distributed workflow management. In *Proc. IFIP Working Conf. on Inf. Syst. for Decentralized Organisations*, Trondheim, Norway.

Barbará, D., Mehrotra, S., & Rusinkiewicz, M. (1996). *INCAs:* Managing dynamic workflows in distributed environments. *Journal of Database Management*, 7(1), 5–15.

Bassil, S., Keller, R., & Kropf, P. (2004). A workflow-oriented system architecture for the management of container transportation, in: *Proc. BPM'04* (pp. 116-131), Potsdam, Germany, LNCS 3080.

Bauer, T., & P. Dadam (1997). A distributed execution environment for large-scale workflow management systems with subnets and server migration. In *Proc. CoopIS'97* (pp. 99–108).

Bauer, T., & Dadam, P. (1999). Efficient distributed control of enterprise-wide and cross-enterprise workflows. In *Proc. GI-Workshop on Enterprise-wide and Cross-enterprise Workflow Management: Concepts, Systems, Applications* (pp. 25–32), Paderborn, Germany.

Bauer, T., & Dadam, P. (2000). Efficient distributed workflow management based on variable server assignments. In *Proceedings CAiSE'00* (pp. 94–109), Stockholm, Sweden.

Bauer, T., Reichert, M., & Dadam, P. (2001). Effiziente Übertragung von Prozessinstanzdaten in verteilten Workflow-Management-Systemen. *Informatik - Forschung und Entwicklung*, 16(2), 76-92.

Bauer, T., Reichert, M., & Dadam, P. (2003). Intra-subnet load balancing for distributed workflow management systems. *International Journal of Cooperative Information Systems*, 12(3), 295–323.

Bauer, T., & Reichert, M. (2004). Dynamic change of server assignments in distributed workflow management systems. In: *Proc. ICEIS'04* (pp. 91-98), Porto, Portugal.

Casati, F., Grefen, P., Pernici, B., Pozzi, H., & Sánchez. G. (1996). *WIDE: workflow model and architecture*. CTIT Technical Report 96-19, University of Twente, The Netherlands.

Casati, F., Ceri, S., Pernici, B., & Pozzi, G. (1998). Workflow evolution. *Data & Knowledge Engineering*, 24(3), 211–238.

Cichocki, A., Georgakopoulos, D., & Rusinkiewicz, M. (2000). Workflow migration supporting virtual enterprises. In *Proceedings BIS'00* (pp. 20–35), Poznán, Poland.

Dadam, P., & Reichert, M. (1998). The ADEPT WfMS Project at the University of Ulm. *Proc. 1st European Workshop on Workflow Management*, Zurich, Switzerland.

Dadam, P., & Reichert, M., eds. (1999). Enterprise-wide and cross-enterprise workflow management: concepts, systems, applications. *CEUR Workshop Proceedings, Vol.* 24.

Dadam, P., Reichert, M., & Kuhn, K. (2000). Clinical workflows - the killer application for process-oriented information systems? In: *Proc. 4th Int'l Conf. on Business Information Systems (BIS'00)* (pp. 36-59), Poznan, Poland.

Dogac, A. et al. (1997). Design and implementation of a distributed workflow management system: METUFlow. In: *Proc. NATO Advanced Study Institute on Workflow Management Systems and Interoperability* (pp. 61–91), Istanbul, Turkey.

Ellis, C.A., & Maltzahn, C. (1997). The Chautauqua workflow system. In *Proc. 30th Hawaii Int. Conf. on System Sciences*, Maui, Hawaii.

Geppert, A., & Tombros, D. (1998). Event-based distributed workflow execution with EVE. In *Proc. IFIP Int. Conf. on Distributed Systems Platforms and Open Distributed Processing* (pp. 427–442).

Golani, M., & Gal, A. (2006). Optimizing exception handling in workflows using process restructuring, In: *Proc. BPM'06* (pp. 407-413), Vienna, Austria, LNCS 4102.

Gray, J., & Reuter, A. (1993). *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers.

Gronemann, B., Joeris, G., Scheil, S., Steinfort, M., & Wache, H. (1999). Supporting cross organizational engineering processes by distributed collaborative workflow management - the MOKASSIN approach. In *Proc. 2nd Symposium on Concurrent Multidisciplinary Engineering*, Bremen, Germany.

Günther, C.W., Rinderle, S., Reichert, M., & van der Aalst, W.M.P. (2006). Change mining in adaptive process management systems. In: *Proc. 14th Int'l Conf. on Cooperative Information Systems (Coopls'06)* (pp. 309-326), Montpellier, France. Springer, LNCS 4275.

Günther, C.W., Reichert, M., & van der Aalst, W.M.P. (2008 a) Supporting flexible processes with adaptive workflow and case Handling. In: *Proceedings WETICE'08, 3rd IEEE Workshop on Agile Cooperative Process-aware Information Systems*, Rome, Italy.

Günther, C.W., Rinderle-Ma, S., Reichert, M., van der Aalst, W.M.P., & Recker, J. (2008 b) Using process mining to learn from process changes in evolutionary systems. *Int'l Journal of Business Process Integration and Management*, *Special Issue on Business Process Flexibility* , 3(1), 61-78

Guth, V., Lenz, K., & Oberweis, A. (1998). Distributed workflow execution based on fragmentation of Petri nets. In *Proc. 15th IFIP World Computer Congress: Telecooperation - The Global Office, Teleworking and Communication Tool* (pp. 114–125).

Han, Y., & Sheth, A. (1998). On adaptive workflow modeling. In *Proc. 4th Int. Conf. on Information Systems Analysis and Synthesisis*, Orlando

Hallerbach, A., Bauer, T., & Reichert, M. (2008). Managing process variants in the process lifecycle. In: *Proc. 10th Int'l Conf. on Enterprise Information Systems (ICEIS'08)*, (pp. 154-161), Barcelona.

Jablonski, S. (1997). Architecture of workflow management systems. *Informatik, Forschung und Entwicklung*, 12(2), 72–81.

Joeris, G., & Herzog, O. (1998). Managing evolving workflow specifications. In *Proceedings CoopIS'98* (pp. 310–321), New York.

Kamath, M., Alonso, G., Günthör, R., & Mohan, C. (1996). Providing high availability in very large workflow management systems. In *Proc. EDBT'96* (pp. 427–442), Avignon, France.

Kochut, K., Arnold, J., Sheth, A., Miller, J., Kraemer, E., Arpinar, B., & Cardoso, J. (2003). IntelliGEN: a distributed workflow system for dscovering protein-protein interactions, *Distributed and Parallel Databases*, 13 (1), 43-72.

Kradolfer, M., & Geppert, A. (1999). Dynamic workflow schema evolution based on workflow type versioning and workflow migration. In *Proc. CoopIS'99* (pp. 104–114), Edinburgh, Scotland.

Lenz, R., & Reichert, M. (2007). IT support for healthcare processes - premises, challenges, perspectives. *Data & Knowledge Engineering*, 61, 82–111.

Li, C., Reichert, M., & Wombacher, A. (2008). Discovering reference process models by mining process variants. In: *Proc. 6th Int'l IEEE Conference on Web Services (ICWS'08)*, Beijing, pp. 45-53

Ly, L.T., Rinderle, S., Dadam, P., & Reichert, M. (2005). Mining staff assignment rules from event-based data. In: *Proc. Workshop on Business Process Intelligence (BPI)* (pp. 177-190), Workshop in conjunction with BPM'05 conference, Nancy, France, LNCS 3812.

Minor, M., Schmalen, D., Koldehoff, A., & Bergmann, R. (2007). Structural adaptation of workflows supported by a suspension mechanism and by case-based reasoning, In: *Proc. WETICE'07* (pp. 370-375), Paris.

Montagut, F., & Molva, R. (2007). Enforcing integrity of execution in distributed workflow management systems, In: IEEE Conf. on Services Computing (SCC'07) (pp. 1-8)

Mouràò, H., & Antunes, P. (2007). Supporting effective unexpected exceptions handling in workflow management systems, In: *Proc. SAC'07* (pp. 1242-1249)

Müller, D., Herbst, J., Hammori, M., & Reichert, M. (2006). IT support for release management processes in the automotive industry. In: *Proc. 4th Int'l Conf. on Business Process Management (BPM'06)* (pp. 368-377), Vienna, Austria, LNCS 4102.

Müller, D., Reichert, M., & Herbst, J. (2007). Data-driven modeling and coordination of large process structures. In: *Proc. 15th Int'l Conf. on Cooperative Information Systems (CoopIS'07)* (pp. 131-149), Vilamoura, Portugal, LNCS 4803.

Müller, D. and Reichert, M., & Herbst, J. (2008) A new paradigm for the enactment and dynamic adaptation of data-driven process structures. In: *Proc. 20th Int'l Conf. on Advanced Information Systems Engineering (CAiSE'08)* (pp. 48-63), Montpellier, France, LNCS 5074.

Muth, P., Wodtke, D., Weißenfels, J., Kotz-Dittrich, A., & Weikum, G. (1998). From centralized workflow specification to distributed workflow execution. *J of Intelligent Inf. Sys.*, 10(2), 159–184.

Mutschler, B., Bumiller, J., & Reichert, M. (2006). Why Process-Orientation is Scarce: An empirical study of process-oriented information systems in the automotive industry. In: *Proc. 10th IEEE Int. Conf. on Enterprise Computing (EDOC '06)* (pp. 433-440), Hong Kong, China.

Mutschler, B., Reichert, M., & Bumiller, J. (2008 a). Unleashing the effectiveness of process-oriented information systems: problem analysis, critical success factors and implications. *IEEE Transactions on Systems, Man, and Cybernetics (Part C),* 38(3), 280-291.

Mutschler, B., Weber, B., & Reichert, M. (2008 b). Workflow management versus case handling: results from a controlled software experiment. In: *Proc. 23rd Annual ACM Symposium on Applied Computing (SAC'08)* (pp. 82-89), Fortaleza, Ceará, Brazil.

Pesic, M., Schonenberg, M., Sidorova, N., van der Aalst, W.M.P. (2007). Constraint-based workflow models: change made easy., in: *Proc. CoopIS'07* (pp. 77-94), Vilamoura, Portugal, LNCS 4803.

Reichert, M., & Dadam, P. (1998 a). ADEPTflex – supporting dynamic changes of workflows without losing control. *Journal of Intelligent Information Systems*, 10(2), 93–129, 1998.

Reichert, M., Hensinger, C., & Dadam, P. (1998 b). Supporting adaptive workflows in advanced application environments. In: *Proc. EDBT Workshop on Workflow Management Systems* (pp. 100-109), March 1998, Valencia, Spain.

Reichert, M., Bauer, T., & Dadam, P. (1999). Enterprise-wide and cross-enterprise workflow management: challenges and research issues for adaptive workflows. In: *Proc. Workshop Informatik '99*, CEUR Workshop Proceedings, Vol. 24 (pp. 56-64), Paderborn, Germany.

Reichert, M. (2000). Dynamische Ablaufänderungen in Workflow-Management-Systemen. *PhD thesis*, University of Ulm (in German).

Reichert, M., Dadam, P., & Bauer, T. (2003 a). Dealing with forward and backward jumps in workflow management systems. *Int'l Journal Software and Systems Modeling*, 2(1), 37-58.

Reichert, M., Rinderle, S., & Dadam, P. (2003 b). On the common support of workflow type and instance changes under correctness constraints. In: *Proc. 11th Int'l Conf. Cooperative Information Systems (CooplS '03)* (pp. 407-425), Catania, Italy, LNCS 2888.

Reichert, M., Rinderle, S., & Dadam, P. (2003 c) ADEPT workflow management system - flexible support for enterprise-wide business processes. In: *Proc. 1st Int'l Conf. on Business Process Management (BPM '03)* (pp. 371-379), Eindhoven, Netherlands, LNCS 2678.

Reichert, M., Rinderle, S., Kreher, U., & Dadam, P. (2005) Adaptive process management with ADEPT2. In: *Proc. Int'l Conf. on Data Engineering (ICDE'05)* (pp. 1113-1114), Tokyo.

Reichert, M., & Bauer, T. (2007): Supporting ad-hoc changes in distributed workflow management aystems. In *Proc. CoopIS'07* (pp. 150 – 168), Vilamoura, Portugal, LNCS 4803.

Rinderle, S., Reichert, M., & Dadam, P. (2003) Evaluation of correctness criteria for dynamic workflow changes. In: *Proc. 1st Int'l Conf. on Business Process Management (BPM '03)* (pp. 41-57), Eindhoven, Netherlands, LNCS 2678,

Rinderle, S., Reichert, M., & Dadam, P. (2004 a). Flexible support of team processes by adaptive workflow systems. *Distributed and Parallel Databases*, 16(1), 91–116.

Rinderle, S., Reichert, M., & Dadam, P. (2004 b). Disjoint and overlapping process changes: challenges, solutions, applications. *In: Proc. 11th Int'l Conf. on Cooperative Information Systems (CooplS'04)* (pp. 101-121), October 2004, Agia Napa, Cyprus. LNCS 3290.

Rinderle, S., Reichert, M., & Dadam, P. (2004 c). On dealing with structural conflicts between process type and instance changes. In: *Proc. 2nd. Int'l Conf. Business Process Management (BPM'04)* (pp. 274-289), June 2004, Potsdam, Germany, LNCS 3080.

Rinderle, S., Weber, B., Reichert, M., & Wild, W. (2005) Integrating process learning and process evolution - a semantics based approach. In: *Proc. 3rd Int'l Conf. on Business Process Management (BPM'05)* (pp. 252-267), Nancy, France. LNCS 3649.

Rinderle, S., Reichert, M., Jurisch, M., & Kreher, U. (2006 a) On representing, purging, and utilizing change logs in process management systems. In: *Proc. 4th Int'l Conf. on Business Process Management (BPM'06)* (pp. 241-256), Vienna, Austria, LNCS 4102.

Rinderle, S., Wombacher, A., & Reichert, M. (2006 b) Evolution of process choreographies in DYCHOR. In: *Proc. 14th Int'l Conf. on Coop. Inf. Sys.* (pp. 273-290), Montpellier, LNCS 4275.

Rinderle-Ma, S., & Reichert, M. (2007). A formal framework for adaptive access control models. *Journal on Data Semantics IX* , Springer, LNCS 4601, 82-112.

Rinderle-Ma, S., Reichert, M., & Weber, B. (2008 a). Relaxed compliance notions in adaptive process management systems. In: *Proc. 27th Int'l Conf. on Conceptual Modeling (ER'08)* (pp. 232-247), Barcelona, Spain, LNCS 5231.

Rinderle-Ma, S., Reichert, M., & Weber, B. (2008 b) On the formal semantics of change patterns in process-aware information systems. In: *Proc. 27th Int'l Conference on Conceptual Modeling (ER'08)* (pp. 279-293), Barcelona, Spain, LNCS 5231.

Rinderle-Ma, S., & Reichert, M. (2008 c). Managing the life cycle of access rules in CEOSIS. In: *Proc. of the 12th IEEE Int'l Enterprise Computing Conference (EDOC'08)* (pp. 257-266), Munich, Germany.

Sadiq, S., Sadiq, W., & Orlowska, M. (2001). Pockets of flexibility in workflow specifications, In: *Proc. ER'01* (pp. 513-526).

Sadiq, S., Sadiq, W., & Orlowska, M. (2005). A framework for constraint specification and validation inflexible workflows, *Information Systems*, 30 (5), 349-378.

Schill, A., & Mittasch, C. (1996). Workflow management systems on top of OSF DCE and OMG Corba. *Distributed Systems Engineering*, 3(4), 206-233

Schuster, H., Neeb, J., & Schamburger, R. (1999). A configuration management approach for large workflow management systems. In *Proc. Int. Conf. on Work Activities Coordination and Collaboration*, San Francisco, 1999.

Shegalov, G., Gillmann, M., & Weikum, G. (2001). XML-enabled workflow management for e-services across heterogeneous platforms. *VLDB Journal*, 10(1), 91–103.

Sheth, A., & Kochut, K.J. (1997). Workflow applications to research agenda: scalable and dynamic work coordination and collaboration systems. In: *Proc. NATO Advanced Study Institute on Workflow Management Systems and Interoperability* (pp. 12–21), Istanbul, Turkey.

Staffware (2003). *Server Administration Guide*. Tool Documentation.

Weber, B., Wild, W., & Breu, R. (2004). CBRFlow - enabling adaptive workflow management through conversational case-based reasoning. In *Proc. ECCBR'04* (pp. 434-448), Madrid, Spain, LNCS 3155.

Weber, B., Rinderle, S., Wild, W., & Reichert, M. (2005 a). CCBR-driven business process evolution. In: *Proc. 6th Int'l Conf. on Case-Based Reasoning (ICCBR'05)* (pp. 610-624), Chicago, LNCS 3620.

Weber, B., Reichert, M., Wild, W., & Rinderle, S. (2005 b) Balancing flexibility and security in adaptive process management systems. In: *Proc. 13th Int'l Conf. on Cooperative Information Systems (CooplS '05)* (pp. 59-76), Agia Napa, Cyprus. LNCS 3760.

Weber, B., Reichert, M., Rinderle, S., & Wild, W. (2006 a). Towards a framework for the agile mining of business processes. In: *BPM'05 Workshop Proceedings* (pp. 191-202), Nancy, LNCS 3812.

Weber, B., Reichert, M., & Wild, W. (2006 b). Case-base maintenance for CCBR-based process evolution. In: *Proc. 8th European Conf. on Case-Based Reasoning (ECCBR'06)* (pp. 106-120), Ölüdeniz, Turkey. LNCS 4106.

Weber, B., Rinderle, S., & Reichert, M. (2007). Change patterns and change support features in process-aware information systems. In *Proc. 19th Int'l Conf. on Advanced Information Systems Engineering (CAiSE'07)* (pp. 574–588), Trondheim, LNCS 4495.

Weber, B., Reichert, M. & Rinderle-Ma, S. (2008). Change patterns and change support features - enhancing flexibility in process-aware information systems. *Data and Knowledge Engineering,* 66(3), 438-466.

Weber, B., & Reichert, M. (2008 b). Refactoring process models in large process repositories., in: *Proc. CAiSE'08* (pp. 124-139), Montpellier, France, LNCS 5074.

Weber, B., Reichert, M., Wild, W., & Rinderle-Ma, S. (2009). Providing integrated life cycle support in process-aware information systems. *Int'l Journal of Cooperative Information Systems (IJCIS)*, 18(1).

Weske, M. (1998). Flexible modeling and execution of workflow activities. In *Proc. 31st Hawaii Int. Conf. on Sys Sciences* (pp. 713–722), Hawaii.

Weske, M. (1999). Workflow management through distributed and persistent CORBA workflow objects. In *Proc. CAiSE'99* (pp. 446–450), Heidelberg, Germany.

Weske, M. (2007). Business process management: concepts, methods, technology, Springer.